Homework Assignment

Mazilu Mihai

April 2024

1 Introduction

https://github.com/MaziluMihai7/Homework-Assignment-.git

2 Problem statement

The Traveling Salesman Problem (TSP) is a classic optimization problem in computer science and operations research. Given a list of cities and the distances between each pair of cities, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city. The objective function varies, but in this case, we aim to minimize the longest distance between two consecutive cities along the route.

3 Pseudocode of the algorithms

Breadth-First Search (BFS):

```
function bfs_tsp(distances):
    min_cost = infinity
    best_path = None
    queue = [(initial_city, [initial_city])]
    while queue is not empty:
        current_city, path = queue.pop(0)
        if length(path) == num_cities:
            cost = calculate_path_cost(path, distances)
            if cost < min_cost:
                min_cost = cost
                best_path = path
        else:
            for city in range(num_cities):
                if city not in path:
                    queue.append((city, path + [city]))
    return best_path
```

Least-Cost (Uniform Cost) Search:

```
function least_cost_search_tsp(distances):
    min_cost = infinity
    best_path = None
    queue = [(0, [0])]  # (cost, path)
    while queue is not empty:
        cost, path = pop_from_priority_queue(queue)
        if length(path) == num_cities:
            if cost < min_cost:
                min_cost = cost
                best_path = path
        else:
            for city in range(num_cities):
                if city not in path:
                    new_cost = max(cost, distances[path[-1]][city])
                    insert_into_priority_queue(queue, (new_cost, path + [city]))
    return best_path
```

A Search:*

```
function a_star_tsp(distances):
    min_cost = infinity
    best_path = None
    queue = [(0, [0])]  # (cost + heuristic, path)
    while queue is not empty:
        cost, path = pop_from_priority_queue(queue)
        if length(path) == num_cities:
            if cost < min_cost:
                min_cost = cost
                best_path = path
        else:
            for city in range(num_cities):
                if city not in path:
                    new_cost = max(cost, distances[path[-1]][city])
                    heuristic_value = calculate_heuristic(path + [city], distances)
                    insert_into_priority_queue(queue, (new_cost + heuristic_value, path + [city]))
    return best_path
```

4 Application outline.

High-Level Architectural Overview:

The application is designed to solve the Traveling Salesman Problem (TSP) using three different algorithms: Breadth-First Search (BFS), Least-Cost (Uniform Cost) Search, and A* Search. The architecture follows a modular design, with distinct modules for each algorithm implementation and utility functions.

Specification of Input Data Format:

The input data consists of a distance matrix representing the distances between each pair of cities. The distance matrix is a square matrix where each element at position (i, j) represents the distance between city i and city j. Additionally, the initial city (starting point) needs to be specified.

Specification of Output Data Format:

The output data is the optimal route that visits each city exactly once and returns to the initial city, along with the corresponding cost of the route. The route is represented as a list of city indices in the order they are visited.

List of Modules:

BFS Module: Contains functions for solving the TSP using Breadth-First Search.
Least-Cost Search Module: Contains functions for solving the TSP using Least-Cost (Uniform Cost) Search.
A Search Module*: Contains functions for solving the TSP using A* Search.
Utility Module: Contains utility functions used across different algorithms, such as calculating path cost and heuristic.
List of Functions:

BFS Module:

bfs_tsp(distances):
Description: Solves the TSP using Breadth-First Search algorithm.
Parameters: distances - Distance matrix representing the distances between each pair of cities.
Return Value: Optimal route as a list of city indices.
Least-Cost Search Module:

least_cost_search_tsp(distances):
Description: Solves the TSP using Least-Cost (Uniform Cost) Search algorithm.
Parameters: distances - Distance matrix representing the distances between each pair of cities.
Return Value: Optimal route as a list of city indices.
A Search Module:*

a_star_tsp(distances):
Description: Solves the TSP using A* Search algorithm.

Parameters: distances - Distance matrix representing the distances between each pair of cities.
Return Value: Optimal route as a list of city indices.
Utility Module:

calculate_path_cost(path, distances):
Description: Calculates the total cost of a given path based on the distance matrix.
Parameters: path - List of city indices representing the route, distances - Distance matrix.
Return Value: Total cost of the path.
calculate_heuristic(path, distances):
Description: Calculates the heuristic value for A* Search.
Parameters: path - List of city indices representing the route, distances - Distance matrix.
Return Value: Heuristic value for the given path.

5 Experiments and evaluation

1. Experiment Setup:

Define different sizes of input instances, ranging from small to large, to test the scalability of the algorithms.
Generate distance matrices with random distances between cities.
Optionally, use predefined distance matrices to test specific scenarios or edge cases.

2. Performance Metrics:

Execution time: Measure the time taken by each algorithm to find the optimal solution.
Solution quality: Compare the cost of the solutions obtained by each algorithm with the known optimal solution (if available).
Scalability: Analyze how the algorithms scale with increasing input size.

3. Experiments:

Small Input Instances:
Use small-sized distance matrices (e.g., 4x4 or 5x5) to test the correctness and efficiency of the algorithms.
Compare the solutions obtained by each algorithm and verify if they match the expected optimal solution.
Measure the execution time of each algorithm and compare.
Medium Input Instances:
Increase the size of the distance matrices (e.g., 8x8 or 10x10) to evaluate the scalability of the algorithms.
Analyze the execution time and solution quality of each algorithm.
Observe if any algorithm exhibits significant performance degradation with larger inputs.
Large Input Instances:
Use larger distance matrices (e.g., 15x15 or 20x20) to further test the scalability.
Focus on the execution time and solution quality of each algorithm.
Determine if any algorithm becomes impractical for large-scale instances.
4. Evaluation:

Analyze the results of the experiments to identify the strengths and weaknesses of each algorithm.
Compare the performance of BFS, Least-Cost Search, and A* Search in terms of execution time and solution quality.
Determine which algorithm performs best under different scenarios (e.g., small vs. large instances, dense vs. sparse distance matrices).
Draw conclusions regarding the suitability of each algorithm for practical applications based on the experimental findings.

5. Optimization Opportunities:

Identify potential areas for optimization within each algorithm (e.g., data structures, heuristics).
Experiment with different optimization techniques to improve the efficiency and scalability of the algorithms.
Re-run experiments to evaluate the impact of optimizations on performance.

6 Conclusions

In conclusion, the TSP is a well-known optimization problem with various solution strategies. The implemented strategies—BFS, Least-Cost Search, and A* Search—provide different trade-offs between optimality and efficiency. While exhaustive strategies guarantee optimal solutions, they may be impractical for large problem instances. Heuristic strategies, such as A* Search, offer a balance between optimality and efficiency, making them suitable for real-world applications. This report demonstrates the implementation and comparison of these strategies for solving the TSP problem.